05｜码流结构：原来你是这样的H264

2021-12-01 李江

《攻克视频技术》 课程介绍 >

讲述：李江
时长 23:41　大小 21.70M

你好，我是李江。

上一节课我们一起讨论了视频编码的基本原理。今天，我们就接着来聊聊视频编码的码流结构，这在视频开发工作中是非常重要的。

视频编码标准其实有很多，比如上一节课讲到的 H264、H265、AV1 等，但原理大同小异，都是预测、变换、量化和熵编码等几个步骤。H264 编码可以说是最常用的编码标准，比较经典，所以这节课我们就以 H264 为例来讲解码流结构。在掌握了这些之后，迁移学习其它编码标准的码流结构也就简单多了。

视频编码的码流结构其实就是指视频经过编码之后得到的二进制数据是怎么组织的，换句话说，就是编码后的码流我们怎么将一帧帧编码后的图像数据分离出来，以及在二进制码

流数据中，哪一块数据是一帧图像，哪一块数据是另外一帧图像。

而我们在工程开发中，需要对编码后的数据进行一些解析，以便用于之后的打包。同时我们在打包时也需要判断当前一帧图像数据它的开头和结尾在哪。这些工作的前提就是我们要清楚如何分析编码码流，那么码流结构到底是怎样的，就是当下的学习重点了。

下面我们就以 H264 编码为基础，分析一下它的码流结构，并看看它在工程中是如何应用的。

H264 的编码结构

这里有一些前置知识我们需要先了解一下。我们先一起来看几个重要的概念吧。它们之间有这样一条线索，你在接下来的学习中可以重点关注一下，对于你记忆它们也是非常有帮助的。

首先，清楚帧类型是图像的基础；其次，GOP 是以其中的 IDR 帧作为分隔点的；最后的 Slice 是我们深入帧内部以后的一个重要概念。整个过程，由浅入深。

帧类型

帧类型相信你在平时的工作中可能已经接触过一部分了，比如说我们可能经常听到视频开发工作者说 I 帧、P 帧之类的。其实在 H264 中，帧类型主要分为 3 大类，分别是 I 帧、P 帧和 B 帧。那么它们之间有什么区别呢？接下来我们就来详细聊聊。

在视频编码原理那节课里面，我们讲过为了减少空间冗余和时间冗余，视频编码使用了帧内预测和帧间预测技术，这些都涉及到帧。所以了解帧的类型是很有必要的。

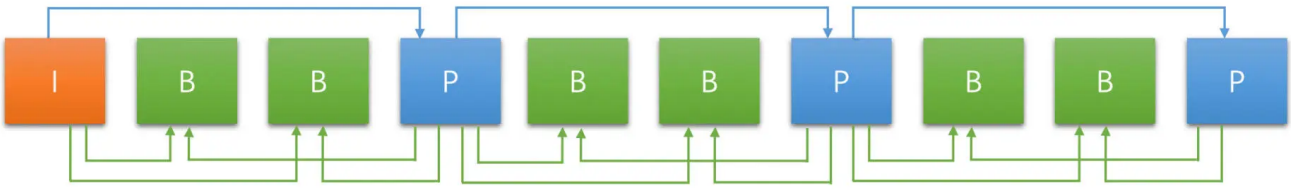
我们知道帧内预测不需要参考已编码帧，对已编码帧是没有依赖的，并可以自行完成编码和解码。而帧间预测是需要参考已编码帧的，并对已编码帧具有依赖性。帧间预测需要参考已经编码好了的帧内编码帧或者帧间编码帧。并且，帧间编码帧又可以分为只参考前面帧的前向编码帧，和既可以参考前面帧又可以参考后面帧的双向编码帧。

为了做区分，在 H264 中，我们就将图像分为以下不同类型的帧。

	帧类型	预测方式	参考帧	优缺点
I帧	帧内编码帧	只进行帧内预测	无	自身能独立完成编码解码。 压缩率小。
P帧	前向编码帧	可以进行帧间预测和帧内预测	参考前面已编码的I帧和P帧	压缩率比I帧高。 必须要参考帧能够正确编解码。
B帧	双向编码帧	可以进行帧间预测和帧内预测	参考前面或者后面已编码的I帧和P帧	压缩率最好。 需要缓存帧，延时高。 RTC场景不适合。

极客时间

三种帧的示例图如下所示。例如，从左向右，第一个 B 帧参考第一个 I 帧和第一个 P 帧，第一个 P 帧只参考第一个 I 帧（箭头是从参考帧指向编码帧）。



由于 P 帧和 B 帧需要参考其它帧。如果编码或者解码的过程中有一个参考帧出现错误的话，那依赖它的 P 帧和 B 帧肯定也会出现错误，而这些有问题的 P 帧（B 帧虽然也可以用来作为参考帧，但是一般用的比较少，所以这里不讨论）又会继续作为之后 P 帧或 B 帧的参考帧。因此，**错误会不断的传递**。为了避免错误的不断传递，就有了**一种特殊的 I 帧叫 IDR 帧**，也叫立即刷新帧。

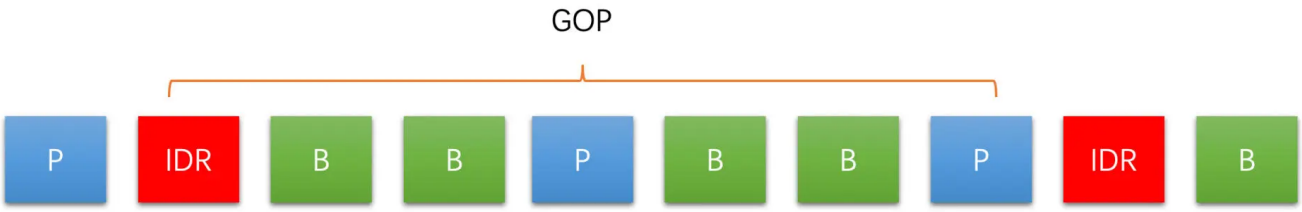
H264 编码标准中规定，**IDR 帧之后的帧不能再参考 IDR 帧之前的帧**。这样，如果某一帧编码错误，之后的帧参考了这个错误帧，则也会出错。此时编码一个 IDR 帧，由于它不参考其它帧，所以只要它自己编码是正确的就不会有问题。之前有错误的帧也不会再被用作参考帧，**这样就截断了编码错误的传递**，且之后的帧就可以正常编 / 解码了。

当然，有 IDR 这种特殊的 I 帧，也就有普通的 I 帧。普通的 I 帧就是指当前帧只使用帧内预测编码，但是后面的 P 帧和 B 帧还是可以参考普通 I 帧之前的帧。但是这里我要说明一下，一般来说我们不太会使用这种普通 I 帧，**大多数情况下还是直接使用 IDR 帧**，尤其是在流媒体场景，比如 RTC 场景。只是说如果你非要用这种普通 I 帧，标准也是支持的。

GOP

在 H264 中，还有一个 GOP 的概念也经常会遇到，它是什么意思呢？从一个 IDR 帧开始到下一个 IDR 帧的前一帧为止，这里面包含的 IDR 帧、普通 I 帧、P 帧和 B 帧，我们称为一个 **GOP（图像组）**（这是 closed GOP，还有一种 opened GOP，比较少见，这里不讨论）。

我们可以看到 GOP 的大小是由 IDR 帧之间的间隔来确定的，而这个间隔我们有一个重要的概念来表示，叫做**关键帧间隔**。关键帧间隔越大，两个 IDR 相隔就会越远，GOP 也就越大；关键帧间隔越小，IDR 相隔也就越近，GOP 就越小。



GOP 越大，编码的 I 帧就会越少。相比而言，P 帧、B 帧的压缩率更高，因此整个视频的编码效率就会越高。但是 GOP 太大，也会导致 IDR 帧距离太大，点播场景时进行视频的 seek 操作就会不方便。

并且，在 RTC 和直播场景中，可能会因为网络原因导致丢包而引起接收端的丢帧，大的 GOP 最终可能导致参考帧丢失而出现解码错误，从而引起长时间花屏和卡顿。这一块我们会在之后用单独的一节课来详细讲述。总之，**GOP 不是越大越好，也不是越小越好，需要根据实际的场景来选择。**

前面我们讲的是视频图像序列的层次结构，那图像内的层次结构是怎样的呢？

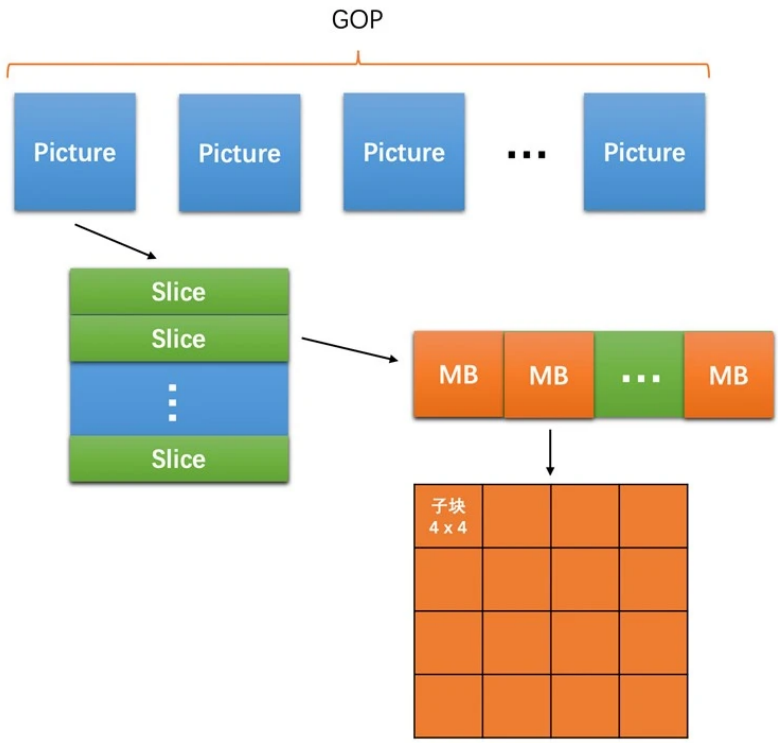
Slice

这就不得不提到另一个概念了，Slice，也叫做“片”。**Slice 其实是为了并行编码设计的。**什么意思呢？就是说，我们可以将一帧图像划分成几个 Slice，并且 Slice 之间相互独立、互不依赖、独立编码。

那么在机器性能比较高的情况下，我们就可以多线程并行对多个 Slice 进行编码，从而提升速度。但也因为一帧内的几个 Slice 是相互独立的，所以如果帧内预测的话，就不能跨 Slice 进行，因此编码性能会差一些。

而在 H264 中编码的基本单元是宏块，所以一个 Slice 又包含整数个宏块。我们在前一节课中也讲了，宏块 MB 大小是 16 x 16。在做帧内和帧间预测的时候，我们又可以将宏块继续划分成不同大小的子块，用来给复杂区域做精细化编码。

总结来说，**图像内的层次结构就是一帧图像可以划分成一个或多个 Slice，而一个 Slice 包含多个宏块，且一个宏块又可以划分成多个不同尺寸的子块**。如下图所示：



好了，上面都是从概念上来讨论视频编码中的视频序列和图像的层次结构。那有了这些知识之后，接下来我们更进一步，从 H264 码流的角度来看看这些层次结构具体在二进制码流中是怎样的。

H264 的码流结构

下面我们就以“剥洋葱”的方式来详细地讲解 H264 的码流结构。先从最外层的码流格式讲起，教你怎么判断视频编码数据的起始；然后再介绍里面的 NALU（网络抽象层单元）数据，看看通过它是怎么区分不同的帧类型的；再详细聊聊 NALU 有几种类型，以及通过什么方式来区分 NALU 的类型。

码流格式

H264 码流有两种格式：**一种是 Annexb 格式**；**一种是 MP4 格式。**两种格式的区别是：

1. Annexb 格式使用**起始码**来表示一个编码数据的开始。起始码本身不是图像编码的内容，只是用来分隔用的。起始码有两种，一种是 4 字节的“00 00 00 01”，一种是 3 字节的“00 00 01”。
   这里需要**注意**一下，由于图像编码出来的数据中也有可能出现“00 00 00 01”和“00 00 01”的数据。那这种情况怎么办呢？为了防止出现这种情况，H264 会将图像编码数据中的下面的几种字节串做如下处理：
   （1）“00 00 00”修改为“00 00 03 00”；
   （2）“00 00 01”修改为“00 00 03 01”；
   （3）“00 00 02”修改为“00 00 03 02”；
   （4）“00 00 03”修改为“00 00 03 03”。

同样地在解码端，我们在去掉起始码之后，也需要将对应的字节串转换回来。

Annexb格式

2. MP4 格式没有起始码，而是**在图像编码数据的开始使用了 4 个字节作为长度标识**，用来表示编码数据的长度，这样我们每次读取 4 个字节，计算出编码数据长度，然后取出编码数据，再继续读取 4 个字节得到长度，一直继续下去就可以取出所有的编码数据了。

MP4格式

这两种格式差别不大，接下来我们主要使用 Annexb 格式来讲解 H264 码流中的 NALU。

下面，我们剥开“洋葱”的最外层，将起始码去掉，进入“洋葱”的内部，也就是编码数据。这个编码数据就是 H264 码流的重要部分——NALU。

NALU

在这节课的开始我们讲了图像分成 I 帧、P 帧和 B 帧这三种类型的帧。其实除了图像数据，视频编码的时候还有一些编码参数数据，为了能够将一些通用的编码参数提取出来，

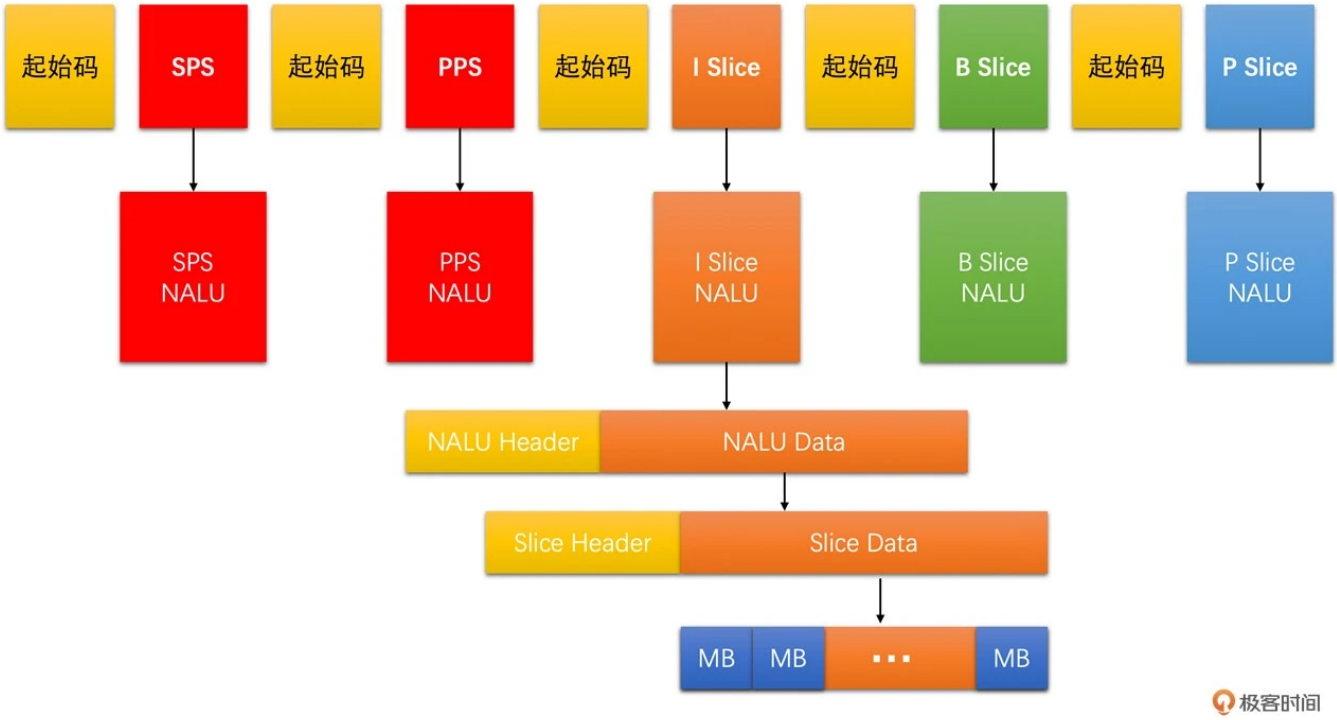
不在图像编码数据中重复，H264 设计了**两个重要的参数集：一个是 SPS（序列参数集）；一个是 PPS（图像参数集）**。

其中，SPS 主要包含的是图像的宽、高、YUV 格式和位深等基本信息；PPS 则主要包含熵编码类型、基础 QP 和最大参考帧数量等基本编码信息。如果没有 SPS、PPS 里面的基础信息，之后的 I 帧、P 帧、B 帧就都没办法进行解码。因此 SPS 和 PPS 是至关重要的。

结合前面我们讲的内容，我们现在可以知道，H264 码流主要包含了 SPS、PPS、I 帧、P 帧和 B 帧。由于帧又可以划分成一个或多个 Slice。因此，帧在码流中实际上是以 Slice 的形式呈现的。所以，**H264 的码流主要是由 SPS、PPS、I Slice、P Slice和B Slice 组成的**。如下图所示：

我们知道了 H264 码流主要由 SPS、PPS 和三种 Slice 组成，那我们**如何在码流中区分这几种数据呢？**

为了解决这个问题，H264 设计了 NALU（网络抽象层单元）。SPS 是一个 NALU、PPS 是一个 NALU、每一个 Slice 也是一个 NALU。每一个 NALU 又都是由一个 1 字节的 NALU Header 和若干字节的 NALU Data 组成的。而对于每一个 Slice NALU，其 NALU Data 又是由 Slice Header 和 Slice Data 组成，并且 Slice Data 又是由一个个 MB Data 组成。其结构如下：



在这里，我们重点介绍一下 NALU Header。它总共占用 1 个字节，具体如下图所示。

0	1	2	3	4	5	6	7
F	NRI			Type			

其中，

F：forbidden_zero_bit，占 1bit，禁止位，H264 码流必须为 0；

NRI： nal_ref_idc，占 2bits，可以取 00～11，表示当前 NALU 的重要性。参考帧、SPS 和 PPS 对应的 NALU 必须要大于 0；

Type： nal_unit_type，占 5bits，表示 NALU 类型。其取值如下表所示。

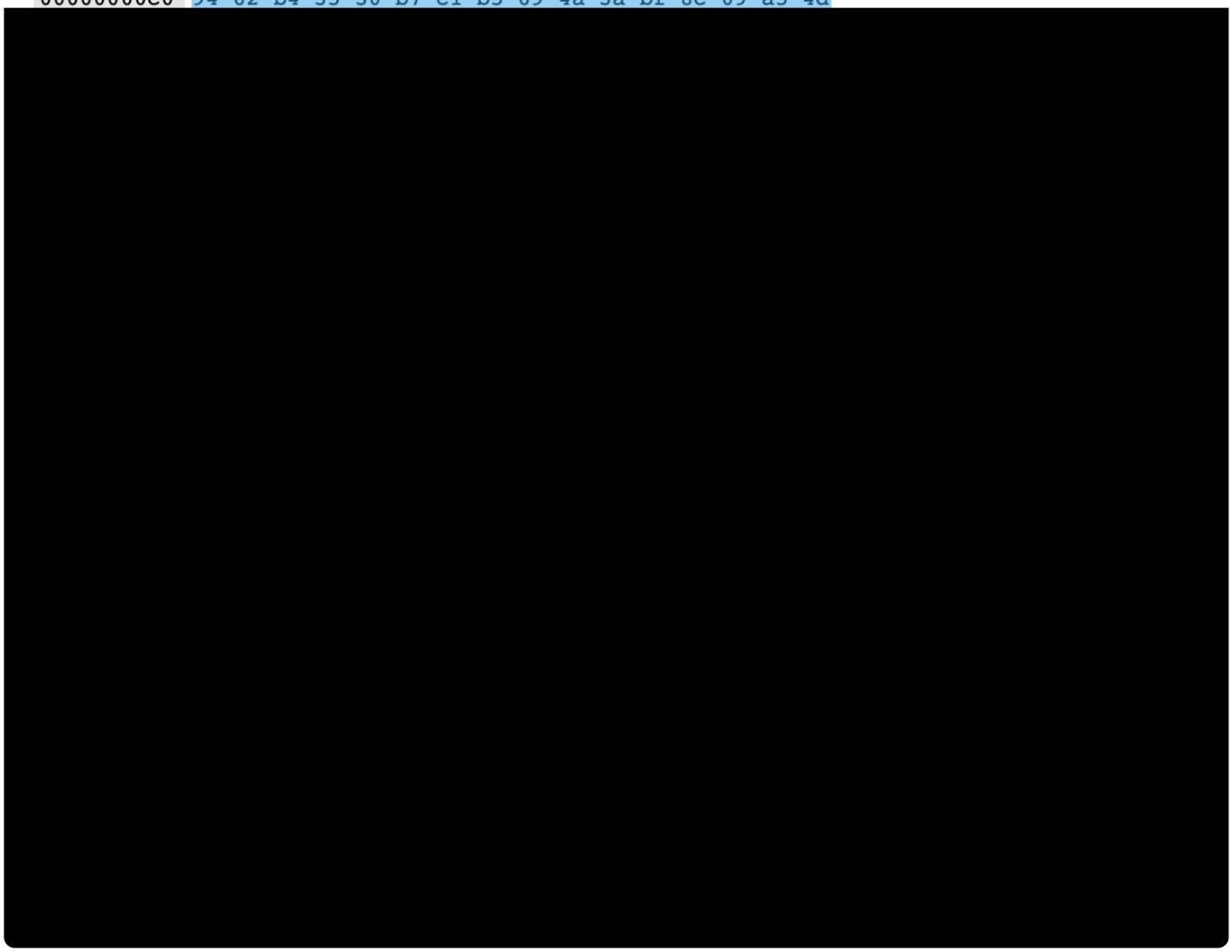
nal_unit_type	0	1	2	3	4	5	6	7	8	9	10	11	12	13…23	24…31
NAL类型	未使用	不分区，非IDR图像的片	片分区A	片分区B	片分区C	IDR图像中的片	补充增强信息单元（SEI）	序列参数集（SPS）	图像参数集（PPS）	分界符	序列结束	码流结束	填充	保留	未使用

有了 NALU Type 类型表格，那我们解析出 NALU Header 的 Type 字段，查询表格就可以得到哪个 NALU 是 SPS，哪个是 PPS，以及哪个是 IDR 帧了。

这里需要注意一下，**NALU 类型只区分了 IDR Slice 和非 IDR Slice**，至于非 IDR Slice 是普通 I Slice、P Slice 还是 B Slice，则需要继续解析 Slice Header 中的 Slice Type 字段得到。我们通过下面两个例子来看看常见的 NALU 里的 NALU Header 是怎样的。

下面我们再来看一个实际码流的例子，看看在实际编码出来的二进制数据中，各种 NALU 是怎么“放置”在数据中的。下图是我用二进制查看工具打开实际编码后的码流数据。我们可以看到在码流的开始部分是一个起始码，之后紧接着是一个 SPS 的 NALU。在 SPS 后面是一个 PPS 的 NALU。然后就是一个 IDR Slice 的 NALU 和一个非 IDR Slice NALU。

```
0000000000 00 00 00 01 67 64 0c 16 ac 19 1a 80 a0 2f 79 60
0000000010 1e 11 08 d4 00 00 00 01 68 ee 3c 80 00 00 00 01
0000000020 65 b8 00 04 00 00 09 ff e5 7e e2 36 c3 b3 23 b5
0000000030 e1 c2 7a 79 80 09 1f 70 d0 fa 1f c9 85 92 90 8f
0000000040 1f ff 57 17 b0 88 35 4e 00 70 f7 07 cc 58 01 98
0000000050 b5 7e 6c dc 41 a0 d0 52 c0 bd ec 73 e9 5b 2d f9
0000000060 8b a0 89 5c 6f e2 e2 0c cf 0d 70 d2 ce 6f af 99
0000000070 b7 2e d4 fa dd 0f 16 2b d6 54 e8 7a 79 da 52 23
0000000080 02 92 af da 40 5e 58 d0 31 f8 36 26 0b b7 56 fc
0000000090 86 5e f5 28 ed 60 c0 98 fe 9e e5 bd 98 36 64 b3
00000000a0 f2 69 46 dc b4 82 29 06 9d 88 49 1b f0 e7 bb b5
00000000b0 af 9c a1 76 2c 7e d8 66 16 80 5a d5 25 e7 1f 68
00000000c0 94 62 b4 35 30 b7 c1 b5 69 4a 3a bf 8c 09 a5 4d
```



现在，对于码流结构的认知你是不是很清晰了。你也可以去找个 H264 码流，用二进制查看工具打开它。通过今天学习的知识，你可以试着找出其中的起始码，看看能不能找到 SPS、PPS、IDR 和非 IDR Slice。如果你都能找出来，那恭喜你说明你已经掌握了今天的主要知识点了。

常见工程问题

好了，在了解了基本的码流结构知识之后，我们来看看如何运用这节课学到的知识去解决工程上常见的一些问题。这里我列举了 3 个比较典型的问题，如果你有更多问题的话，可以到留言区我们一起讨论。

多 Slice 时如何判断哪几个 Slice 是同一帧的？

我们前面讲过，在 H264 码流中，帧是以 Slice 的方式呈现的，或者可以说在 H264 码流里是没有“帧“这种数据的，只有 Slice。但是有个问题是，一帧有几个 Slice 是不会告诉你的。也就是说码流中没有字段表示一帧包含几个 Slice。既然没有办法知道一帧有几个 Slice，那我们如何知道多 Slice 编码时一帧的开始和结束分别对应哪个 Slice 呢？

其实，Slice NALU 由 NALU Header 和 NALU Data 组成，其中 NALU Data 里面就是 Slice 数据，而 Slice 数据又是由 Slice Header 和 Slice Data 组成。**在 Slice Header 开始的地方有一个 first_mb_in_slice 的字段**，表示当前 Slice 的第一个宏块 MB 在当前编码图像中的序号。我们只要解析出这个宏块的序号出来，

> 如果 first_mb_in_slice 的值等于 0，就代表了当前 Slice 的第一个宏块是一帧的第一个宏块，也就是说当前 Slice 就是一帧的第一个 Slice。
>
> 如果 first_mb_in_slice 的值不等于 0，就代表了当前 Slice 不是一帧的第一个 Slice。并且，使用同样的方式一直往下找，直到找到下一个 first_mb_in_slice 为 0 的 Slice，就代表新的一帧的开始，那么其前一个 Slice 就是前一帧的最后一个 Slice 了。

slice_header( ) {	C	描述符
first_mb_in_slice	2	ue(v)
slice_type	2	ue(v)
pic_parameter_set_id	2	ue(v)
frame_num	2	u(v)
if(!frame_mbs_only_flag) {		
field_pic_flag	2	u(1)
if(field_pic_flag)		
bottom_field_flag	2	u(1)
}		
if(nal_unit_type == 5)		
idr_pic_id	2	ue(v)
if(pic_order_cnt_type == 0) {		
pic_order_cnt_lsb	2	u(v)
if(pic_order_present_flag && !field_pic_flag)		
delta_pic_order_cnt_bottom	2	se(v)
}		
if(pic_order_cnt_type == 1 && !delta_pic_order_always_zero_flag) {		
delta_pic_order_cnt[0]	2	se(v)
if(pic_order_present_flag && !field_pic_flag)		
delta_pic_order_cnt[1]	2	se(v)
}		

图片来源于H264标准文档

其中，first_mb_in_slice 是以无符号指数哥伦布编码的，需要使用对应的解码方式才能解码出来。但是有一个小技巧，如果只是需要判断 first_mb_in_slice 是不是等于 0，不需要计算出实际值的话，只需要通过下面的方式计算就可以了。

如果slice_header[0] & 0x80等于1，则first_mb_in_slice为0。

这就是多 Slice 判断一帧的开始和结束的方法。

如何从 SPS 中获取图像的宽高？

在编码端编码一个视频的时候，我们是需要设置分辨率告诉编码器图像的实际宽高的。但是解码器是不需要设置分辨率的，那我们在解码端或者说接收端如何知道视频的分辨率大小呢？

其实，在编码器编码的时候会将分辨率信息编码到 SPS 中。**在 SPS 中有几个字段用来表示分辨率的大小。我们可以解码出这几个字段并通过一定的规则计算得到分辨率的大小。**这几个字段分别是：

字段	意义
pic_width_in_mbs_minus1	图像宽度方向有多少宏块
pic_height_in_map_units_minus1	图像高度方向有多少宏块
frame_cropping_flag	需不需要裁剪
frame_crop_left_offset	左边裁剪大小
frame_crop_right_offset	右边裁剪大小
frame_crop_top_offset	上边裁剪大小
frame_crop_bottom_offset	下边裁剪大小

极客时间

这几个字段都是通过无符号指数哥伦布编码的，需要先解码出来。解码得到具体值之后，通过以下方法就可以得到分辨率了。注意，pic_height_in_map_units_minus1 需要考虑帧编码和场编码的区别，其中场编码已经很少使用了，我们这里不再考虑。

1、如果frame_cropping_flag为0：

width = (pic_width_in_mbs_minus1 + 1) * 16

height = (pic_height_in_map_units_minus1 + 1) * 16

2、如果frame_cropping_flag为1:

width = (pic_width_in_mbs_minus1 + 1) * 16 – (frame_crop_left_offset << h_shift) – (frame_crop_right_offset << h_shift)

height = (pic_height_in_map_units_minus1 + 1) * 16 – (frame_crop_top_offset << v_shift) – (frame_crop_bottom_offset << v_shift)

其中，h_shift在YUV420和YUV422时为1，其余为0；

v_shift在YUV420时为1，其余为0。

极客时间

通过上面的方法就可以计算得到图像的分辨率了。

如何计算得到 QP 值？

我们在视频编码原理那节课中讲过，量化过程是引入失真最主要的环节。而量化最主要的参数就是 QP 值，并且 QP 值的大小严重影响到编码画面的清晰度。因此 QP 值非常重要。那么我们如何从码流中计算得到 QP 值呢？

在 PPS 中有一个全局基础 QP，字段是 pic_init_qp_minus26。当前序列中所有依赖该 PPS 的 Slice 共用这个基础 QP，且每一个 Slice 在这个基础 QP 的基础上做调整。在 Slice Header 中有一个 slice_qp_delta 字段来描述这个调整偏移值。更进一步，H264 允许在宏块级别对 QP 做更进一步的精细化调节。这个字段在宏块数据里面，叫做 mb_qp_delta。

字段	意义
pic_init_qp_minus26	PPS初始基础QP
slice_qp_delta	Slice QP偏移量
mb_qp_delta	宏块QP偏移量

极客时间

如果需要得到 Slice 级别的 QP 则只需要考虑前两个 QP 相关字段。如果需要计算宏块 QP，则需要三个都考虑。但是宏块 QP 需要解析整个 Slice 数据，计算量大。一般我们直接计算到 Slice QP 就可以了。计算方法如下：

Slice级QP：

SliceQP = (pic_init_qp_minus26 + 26) + slice_qp_delta

宏块级QP：

MBQP = (pic_init_qp_minus26 + 26) + slice_qp_delta + mb_qp_delta

小结

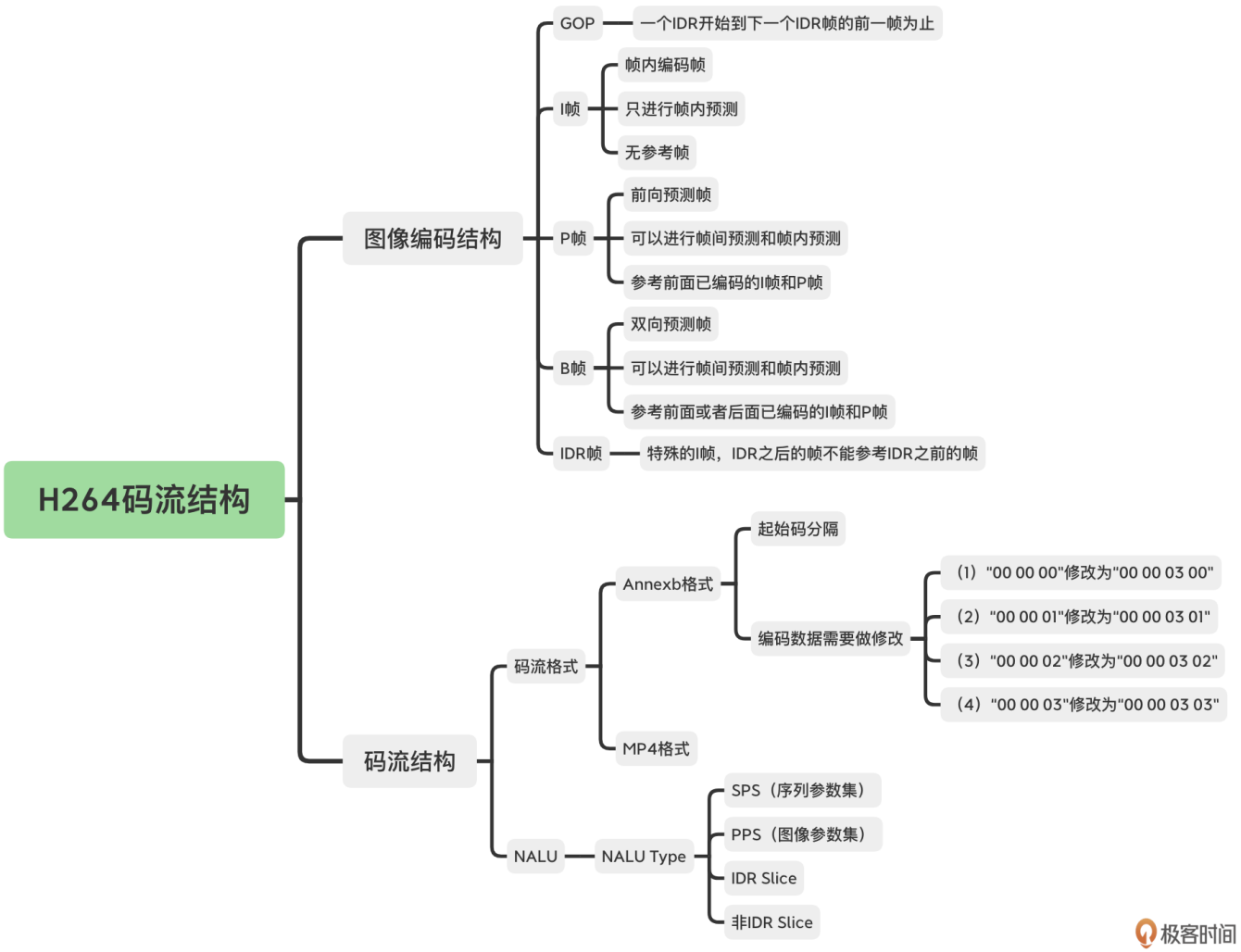
这节课我们主要讨论了 H264 的编码层次结构和码流结构。在一个视频图像序列中，我们将其划分成一个个 GOP。**GOP 包含一个 IDR 帧到下一个 IDR 帧的前一帧中的所有帧。**

GOP 的大小选择需要根据实际应用场景来选择，**一般 RTC 和直播场景可以稍微大一些，而点播场景一般小一些。**

在 H264 中，每一帧图像又可以分为 I 帧、P 帧和 B 帧，而 I 帧又包含了普通 I 帧和 IDR 帧。帧可以划分为一个或者多个 Slice，并且**最后帧都是以 Slice 的方式在码流中呈现**。同时 H264 码流中除了 Slice 数据之外，还有 **SPS 和 PPS 两个参数集，分别用来存放基础图像信息和基础编码参数**。SPS 和 PPS 非常重要，如果丢失了，将无法进行解码。

每一个 Slice 和 SPS、PPS 都是通过 NALU 来封装的，且 NALU 含有一个 1 字节的 NALU Header。我们可以**通过 NALU Header 中的 NALU Type 来判断 NALU 的类型**。同时，每一个 NALU 的分隔有两种方式：一种是 **Annexb 格式，通过使用起始码分隔**；一种是 **MP4 格式，通过一个 4 字节的长度来表示 NALU 的大小**，从而起到分隔的作用。

为了帮助你记忆，我们通过下图来总结一下。



思考题

为什么有 B 帧的时候延时会高？

你可以把你的答案和思考写下来，分享到留言区，与我一起讨论。下节课再见。

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 1　提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇　04 | 编码原理：视频究竟是怎么编码压缩的？

下一篇　06 | 帧内预测：如何减少空间冗余？

精选留言 (16)

写留言

smart
2021-12-01

B帧需要双向参考，pts 和 dts 不一致。因此需要等待后面的 p 帧解码后才能继续，从而引入了延时。

作者回复: 嗯嗯，编码的时候也是一样的，需要先等后面的P帧先编码才能编码B帧。

2

qingzhu
2021-12-02

（1）“00 00 00”修改为“00 00 03 00”；
会不会原始编码中也存在“00 00 03 00”的冲突？

1

晓龙
2021-12-02

想问一下老师，起始码和NALU的关系是，起始码是用来区分PPS，SPS，I，B，P帧，然后I，B，P帧内部，又有由多个Slice/NALU组成是吧？ 也就是说，I 帧和B帧中间用起始码分割，然后I帧内部，又会再被Slice分割？

展开

1

aikeke

2021-12-02

老师能推荐下好用的h264字段分析工具吗？

展开

作者回复: Elecard或者VQAnalyzer都挺好的，我用的VQAnalyzer。

1

qinsi

2021-12-04

希望能知道码流结构为什么要这样设计？上节课讲编码原理的时候，预测编码、DCT和量化等都是为了压缩掉各种冗余信息。这节课的码流结构又解决了什么样的问题呢？尤其是NALU，为什么是这样的结构？有没有什么特别的考虑？还是说只要包含了必要的信息，随便什么样的格式都可以？

展开

我有一条鱼

2021-12-03

问题：
那B帧相对于P帧来说能减少多少大小吗？有相关的数据吗？

展开

qingzhu

2021-12-02

怎么判断数据量是MP4格式的，还是Annexb起始码格式的？

Y

2021-12-02

请问老师：

IDR之前的帧，会参考本IDR帧和之后的帧吗？

ripple
2021-12-02

防止竞争中的02,03是怎么存在的？

展开

晓龙
2021-12-02

iOS Video Toolbox 可以参考这一篇 https://www.cnblogs.com/tangyuanby2/p/11449460.html， 对照老师的课程，融会贯通

展开

作者回复: 嗯嗯，一起学习交流，又问题可以给我留言。

晓龙
2021-12-02

iOS Video ToolBox 可以参考这一篇，在针对老师的课程，融会贯通

晓龙
2021-12-02

因为B帧的解析依赖前后帧，所以需要缓存帧，才能解析B帧，一般RTC场景不编码B帧

作者回复: 回答的非常对。

paradise
2021-12-02

B帧因为双向参考的缘故，编码和解码都存在时延，特别是在倒放的时候比较麻烦，所以安防领域都不使用B帧编码。
感谢江哥细致的讲解

作者回复: 嗯嗯，谢谢支持。

_zeta
2021-12-01

MP4包格式和RTP包格式有什么区别么？

展开

作者回复: 两种叫法。是一样的。

Geek_7de4c5
2021-12-01

老师，请问一下，如何通过坐标来找对应的宏块呢？

展开

作者回复: 是说在码流中去找到宏块？还是图像里面每一个宏块的坐标是怎样的？
如果说在码流中去找到每个宏块的话就比较难了，只有Slice的第一个宏块的序号是可以通过slice_header中的first_mb_in_slice得到，其他的需要一个一个位的进行熵解码出出来才能知道当前解码的宏块是第几个宏块，所以这也是为什么在计算宏块级QP时，需要将一帧都熵解码出来，比较麻烦。
如果只是说图像中宏块的坐标怎么表示的话，就是从左到右、从上到下依次排列的。最左上角的是0号宏块，之后每个宏块序号+1，知道最右下角的宏块。

西钾钾
2021-12-01

B帧是双向预测帧，需要前后帧作为参考，这会在拉流侧产生延迟。
B解码时应该也会比较耗费cpu

展开

作者回复: 编码的时候，由于需要先编码后面的P帧，才能编码前面的B帧，所以也会缓存几帧才能开始编码B帧，也会导致延时很多。